Teaching Parallel Programming for Time-Efficient Computer Applications

A. Asaduzzaman
EECS Department
Wichita State University
Wichita, Kansas

R. Asmatulu
Mech. Eng. Department
Wichita State University
Wichita, Kansas

M. Rahman
School of Comp. Sci. & Eng.
Georgia Institute of Technology
Atlanta, Georgia

ABSTRACT
Academic research and engineering challenge both require high performance computing (HPC), which can be achieved through parallel programming. The existing curricula of most universities do not properly address the major transition from single-core to multicore systems and sequential to parallel programming. They focus on applying application program interface (API) libraries and open multiprocessing (OpenMP), message passing interface (MPI), and compute unified device architecture (CUDA)/GPU techniques. This approach misses the goal of developing students' long-term ability to solve real-life problems by 'thinking in parallel'. In this article, a novel approach is proposed to teach parallel computing that will prepare computer application developers for present and future computation challenges. Using multicore/manycore architecture and popular challenging problems from areas like computer science, proposed approach teaches how to analyze and develop efficient solutions for the problems. As preliminary work, some multithreaded parallel programs are introduced to computer science and engineering students. Based on the feedbacks from information technology (IT) professionals and Student Outcomes Assessment Reports, proposed approach has potential to provide adequate knowledge so that students can fulfill the growing industry demands for HPC. Based on the Steady State Heat Equation experiment, CUDA/GPU parallel programming may achieve up to 241x speed up factor while simulating heat transfer on a 5000x5000 thin surface.

General Terms
Concurrent Processing; GPU Computing; Graphics and Imaging; Principle of Concurrency;

Keywords
CUDA/GPU technology; multicore architecture; OpenMP; Open MPI; parallel programming;

1. INTRODUCTION
Current and future processors are expected to have multiple cores in their CPUs. (Only exception may be some small embedded processors for specialized devices.) Moreover, attached GPU cards with large numbers of cores have become very attractive for high performance computing and can provide orders of magnitude speedup over using the CPU alone [1, 2]. To address this space, Intel has rolled out its Many Integrated Core (MIC) architecture [3]. Systems with a small number of cores such as present multicore processors can use a shared memory model whereas as the number of cores increase, hierarchical user-managed memory and distributed memory models can be expected. Undergraduate programming has yet to address this major transition from single core processors to multicore and many-core processors properly. Training students in this technology is critical to the future of exploiting new computer systems [4]. Today, with all the advances in hardware technology, the educators find themselves with multicore computer as servers, desktops, personal computers, and even handheld devices in the laboratories while still teaching undergraduate students how to design system software, algorithms and programming languages for sequential environment [5]. The current practice is to introduce parallel programming at graduate-level (only at some high-ranked universities), starting with parallel libraries – OpenMP and thread APIs for shared memory systems [6], MPI for message-passing distributed memory systems [7], and CUDA/C for high performance GPU computing [1]. Usually, a course will begin with learning a library, typically MPI applied to a simple parallel applications such as matrix multiplication or sorting, then move onto thread-based tools such as OpenMP, and finally onto programming GPUs with multithreaded CUDA/C [7-11]. The focus is on learning programming libraries applied to a few simple parallel applications. This approach does not fulfill the goal of developing more long-term abilities to reason about parallel solutions and solve larger problems for multiprocessor systems. However, the demands for parallel programmers in the industries are increasing. Based on an insidePHC report, from November 2009 to July 2011 CUDA jobs increased 22%, OpenMP jobs increased 85%, and MPI jobs increased 33% [12]. Therefore, an approach to teach parallel programming is needed that focuses on higher-level programming strategies for computational problems and especially on ease of programmability [13].

The rest of the paper is organized as follows: Section 2 presents the proposed approach to develop/update pedagogy for teaching parallel programming. Learning materials are discussed in Section 3. Section 4 summarizes the course overview. In Section 5, some preliminary work is discussed as examples of CUDA/GPU assisted multithreaded parallel programming model. Finally, this work is concluded in Section 6.

2. PROPOSED APPROACH
2.1 Major Steps
This proposal includes right-to-the-industry-needs activities to prepare students for future computational engineering challenges. Major steps to develop a new pedagogy or update an existing pedagogy are shown in Figure 1. The proposed approach has four major steps: Analysis, development, implementation, and assessment.

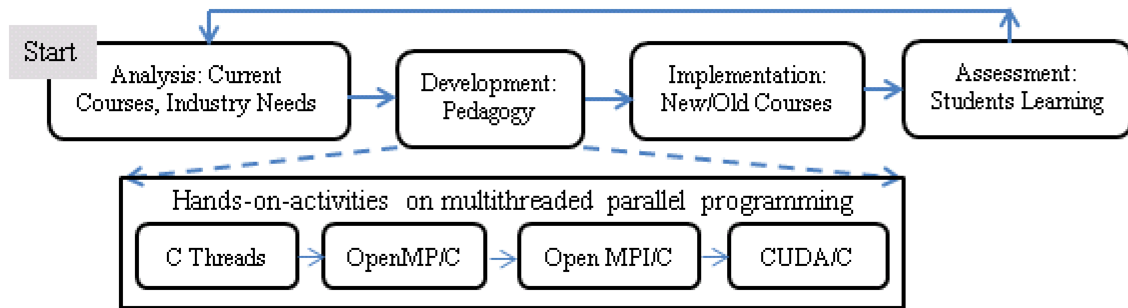

Fig 1: Top line: pedagogy development and integration with existing course(s). Bottom line: hands-on-activities based on real-world IT needs using multicore parallel programming.

First, industry needs and current courses are probed to determine if a new course is needed or existing courses should be updated. Then, pedagogy is developed and implemented (accordingly). Finally, students' outcomes are assessed. Feedbacks from students and industry-professionals are considered to improve the existing pedagogy.

It is envisioned that in order to fulfill the growing IT industry needs, multicore parallel programming will be made available to all undergraduate/graduate engineering students by updating and restructuring existing courses (rather than introducing new courses) [14, 15]. However, this paper provides a complete documentation to prepare a new course or update existing course(s).

2.2 Involving High School and College Educators

It is often noticed that the students have fear about science, technology, engineering, and mathematics (STEM) courses. One reason may be that the STEM education, especially the new technology, is not effectively transformed to the students during their high-school and/or lower-level college/university years. This causes serious problems for educators to teach and students to learn upper-level undergraduate and graduate level courses. Therefore, it is important to involve local high school, college, and university teachers to discuss and address how to improve STEM education and students' learning. Each workshop should help review the current progress and determine the future adjustments.

3. LEARNING MATERIALS

The importance of developing a successful strategy to teach parallel computing and programming has been raised many times over and over. Peter Pacheco designed and offered a sophomore-level undergraduate/graduate parallel computing course in the Department of Computer Science and Mathematics at the University of San Francisco for the first time in 2004. One of the major goals of the course was to provide the students with hands-on experience and encourage them to start to think in parallel. His recommendation was "don't expect the students to discover how to write parallel programs: give them a lot of guidance." Given that the goal of the course is to help students to "think in parallel", the environment should be provided within which students solve problems with parallelism as default.

Programming multicore computers with shared memory programming languages will be focused as well as on message passing programming environments for this course. The foundations for thinking in parallel can be better built within the scope of shared memory. The MPI programming and design developed for message passing distributed platforms adds an additional level of complexity and challenge to problem, data, and program partitioning that can be further explored as an advanced level. This experience was shared by Adams, Nevison, and Schaller who designed three different parallel computing courses at three different colleges, Calvin, Colgate, and RIT [16].

In many cases, the best parallel solution will perform poorly on a sequential machine. The parallel solution performs better only when it is executed in parallel on a parallel computer with enough number of processors. Learning about the trade-offs between parallelism and memory usage, inherently sequential access data structures versus data structures that allow for parallel access, and allowing more operations to be performed in a parallel version compared to the sequential version solving the same problem can be done most effectively when students observe these factors in a hands-on laboratory environment.

4. COURSE OVERVIEW

4.1 Grading Policy

There should be about 10 homework, 4 quizzes, 2 exams, and 1 team-project. Types and points distribution for various activities are shown in Table 1. Some additional information like day/time is also suggested in the table.

Table 1. Grading Policy: activities and points

Activity (No.)		Point	Description
Homework (10)		10%	Take home assignment
Quiz (4)		20%	Classroom, closed-book
Mid-Term (1)		25%	Classroom, closed-book
Final (1)		25%	Classroom, closed-book
Team-Project	Survey/ Proposal	5%	Project proposal per group
	Demo/ Presentation	6%	Poster/PPT slides presentation per group
	Final Report	9%	15+ pages per group

4.2 Course Outline

It is expected that students in this class have introductory knowledge on computer architecture and programming in C/C++. A high-level course outline of the proposed semester long senior-level course is presented next.

4.2.1 Module 1: Background and Motivation

This module will introduce the parallel computing by means of evolution of parallelism, concurrency, and multicore computer architectures with specific examples to demonstrate each concept.

4.2.2 Module 2: Observing Parallelism
This module introduces the data dependence relationships and their impact on the ability to perform parallel operations using dataflow graphs. Performance analysis for these computations will be presented using size, depth, speedup and efficiency of algorithms. Graph theory will be introduced and applied to several example computations.

4.2.3 Module 3: Getting Started
To express algorithms, a set of pseudo code conventions for expressing sequential and basic parallel operations such as process creation and termination (fork/join) and storage classes (shared/private variables) will be presented. An example parallel pseudo code such as matrix multiplication will illustrate the parallel operations. Pthread in C/C++ will be used to illustrate multithreaded programming.

4.2.4 Module 4: Programming Shared Memory Multicore Computers
In this module, OpenMP is presented within the context of solving some numerical algorithms. The key to this section is to start programming and present the examples in the context of global parallelism (think in parallel) [17]. Lectures will cover the basic operations on dense matrices such as matrix multiplication to introduce storage layout and various parallel loops.

4.2.5 Module 5: Trivial Parallelism
In this module, a set of interesting but easy to parallelize problems like image processing will be introduced. These problems require the simplest parallel solution whose computation can obviously be divided into a number of completely independent parts, each of which can be executed by a separate processor/core. This module introduces Open MPI for shared and distributed memory parallel programming.

4.2.6 Module 6: Massive Parallelism
Graph algorithms will be solved using CUDA/GPU based parallel programming technique. Many massive and/or complex problems (Prim's minimum spanning tree algorithm) can be expressed in terms of graphs, and can be solved using standard graph algorithms. Students will learn how to decompose a graph into sub-graphs with the goal of optimizing load balance and minimizing synchronization overhead. GPU-shared-memory programming will also be covered in this module.

4.2.7 Module 7 (optional; if time allows): Sorting Algorithms
A number of different types of parallel sorting schemes have been developed for a variety of parallel computer architectures [18-20]. The lectures of this topic present several parallel sorting algorithms such as merge sort, quicksort, bucket sort, and bitonic sort.

4.2.8 Team Project Ideas
Any problem that can be solved by writing computer programs and/or developing computer simulations but takes significant amount of time will be considered as a nice project topic for this course. Several typical applications from different domains are selected that can be parallelized for possible team projects. Students are welcome to propose their own project ideas for approval. Some team-project examples are described below:

Lightning Strike Protection on Nanocomposites: The lack of lightning strike protection (LSP) for the nanocomposite materials limits their use in many applications including aircrafts. As a result, there is a continuous interest in understanding the heterogeneous thermoelectric behavior of mixtures with carbon fibers/nanotubes of these materials. Currently available methodologies, including computer simulation, to assess the thermoelectric behavior of composite materials are extremely time consuming, expensive, and ineffectual. A fast and effective simulation model can be developed using CUDA/GPU technology to analyze LSP on nanocomposite aircrafts.

Processing Large Images: Processing large images is computing intensive and time consuming. Applying various image filters through the GPU parallel programming should improve the overall performance while processing larger images without compromising the existing resources. Other parallel programming techniques like OpenMP and Open MPI can also be used.

Deterministic Primality Test: Prime numbers play an important role in maintaining the secret spy codes. Computer hackers try to steal information or break into private transactions. Computer security authorities use extremely large prime numbers when they devise cryptographs for protecting vital information that is transmitted between computers [21]. The primality test on GPU is expected to be faster than on CPU for large numbers, such as those used in public key cryptography. Using parallel solutions (like OpenMP, Open MPI, and CUDA/GPU) not only should save time, but also should reduce power consumption.

Improve Decryption in a Partially Homomorphic Encryption Schemes: In cryptography, public key algorithms are widely known to be slower than symmetric key alternatives for their basis in modular arithmetic. The modular arithmetic in RSA (for R. Rivest, A. Shamir and L. Adleman; 1977) [21] and Diffie Hellman is computationally heavy when compared to symmetric algorithms relying on simple operations like shifting of bits and XOR. Parallel techniques (like OpenMP, Open MPI, and CUDA/GPU) can be used to make a more efficient and faster implementation of public key algorithms.

5. PRELIMINARY WORK
Three pieces of work, Steady-State Heat Equation for thermal conductivity, Laplace’s Equation for electric charge distribution, and Convolution for image processing, are discussed to illustrate the potential of multithreaded parallel programming using CUDA/GPU technology. However, at first the computing systems used in this study are introduced.

5.1 Computing Platform
5.1.1 Software Used
Linux Debian 7.0 operating system and GNU Compiler Collection (GCC) version 4.6.3 are used. CUDA is configured following the instructions provided in NVIDIA Developer Zone (URL: https://developer.nvidia.com/cuda-downloads). NVIDIA developed the parallel computing platform and programming model CUDA to program GPUs [21, 22]. CUDA provides access to the virtual instruction set and memory of CUDA GPUs. CUDA makes GPUs accessible for computation like CPUs. GPUs have a parallel throughput architecture that emphasizes executing many concurrent threads slowly, rather than executing a single thread very quickly. This approach of solving general-purpose problems on GPUs is known as GPU. The CUDA platform is accessible to software developers through CUDA-accelerated libraries, compiler directives (such as OpenACC), and extensions to

industry-standard programming languages, including C, C++, and Fortran.

5.1.2 Hardware Used

The simulation models in a CUDA server, CPU with general-purpose GPU (GPGPU) card, are run. Important parameters of the CPU and the GPU are listed in Table 1.

Table 2. Important System Parameters

CPU	GPGPU
• Processor: Intel Xeon E5506 • Cores: 2 x Quad-Core • Clock Speed: 2.13 GHz • RAM: 8GB DDR3 • Max. Memory Bandwidth: 19.2 GB/sec • OS: Linux (Debian)	• Type: NVIDIA Tesla C2075 • Cores: 14 x 32 Cores • RAM: 6GB GDDR5 • RAM Speed: 1.5 GHz • RAM Bandwidth: 144 GB/sec • OS: Not applicable

5.2 Solving Steady State Heat Equation

First, a CUDA accelerated parallel programming technique is presented to solve steady state heat equation [23, 24]. Let's consider the heat flow in a one-dimensional uniform bar. If two nearby points on the rod, separately by a small distance d are at temperatures t1 on the left and t2 on the right, then the heat flow from left to right between these points is proportional to the temperature difference and inversely proportional to the distance as shown in Equation 1.

Amount of heat per unit time = k(t1 – t2)/d (1) ………… (1)

Where, the constant of proportionality k is the thermal conductivity and it depends only on the materials that make up the rod. Now, the discrete approach of heat conduction on a 2D surface is explained. Consider a physical region (width w * height h) that is covered with a grid of m * n nodes (see Figure 2). An m * n array A is used to record the temperature of each node. The correspondence between array indices and locations in the region is suggested by giving the indices of the four corners:

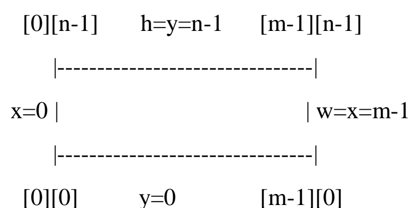

Fig 2: A physical region with boundary conditions

The steady state solution to the discrete heat equation satisfies the following condition (see Equation 2) at an interior grid point:

A [x, y] = (1/4) * (A [x-1, y] + A [x+1, y] + A [x, y+1] + A [x, y-1]) …………………………………………………… (2)

Where, [x, y] is the index of the grid point, [x-1, y] is the index of its immediate neighbor to the "left/west", and so on. Given an approximate solution of the steady state heat equation, a "better" solution is given by replacing each interior point by the average of its 4 neighbors – i.e., by using the condition as an assignment statement (see Equation 3):

A [x, y] <= (1/4) * (a [x-1, y] + A [x+1, y] + A [x, y+1] + A [x, y-1]) …………………………………………………… (3)

If this process is repeated often enough, the difference between successive estimates of the solution will go to zero (or close to zero). In the main loop, after calculating each new A[x, y] value, it is checked if the value is in the acceptable range or not. This approach is used in the experiments. However, using parallel programming like CUDA/C, the sequential loop into parallel equivalent threads is converted and run them concurrently on the GPU cores. The main loop in CUDA/C is shown in Figure 3.

```
/* The new estimate W is the average of north, south, east,
and west neighbors */
diff = 0.0;
for ( i = 1; i < M - 1; i++ ) {
for ( j = 1; j < N - 1; j++ ) {
w[i][j] = (u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1]) / 4.0;
/* determine if more iteration will be repeated or not */
if ( diff < fabs ( w[i][j] - u[i][j] ) ) {
diff = fabs ( w[i][j] - u[i][j] );
}}
}/* end for...i */
```

Fig 3: Steady State Heat Equation (main loop in C)

To implement the steady state heat equation (as shown in Equation 3) on a 2D surface, it is consider that an n * n very thin metal surface has N * N nodes; where N = 100, 500, 1000, 2500, or 5000. Initially, all the boundary nodes (where x=0, y=0, x=n-1, or y=n-1) are given a value of 0.00 (these values do not change). Also initially, any one node (x, y), where $1 \leq x \leq n-1$ and $1 \leq y \leq n-1$, is assumed to have a very high value (1000000 in the experiment). Then the new values for all nodes are calculated. These iterations are repeated until the new value of a node becomes less than a predefined small value, often called 'error tolerance' (0.0001 in the experiment). Experimental results (CPU time and GPU time) are shown in Table 2. First thing to notice is that both the CPU time and the GPU time increase significantly as the problem size increases. At the beginning, for smaller problem size, CPU time is actually less than the GPU time. However, as the problem size keeps getting bigger, the GPU time keeps getter better (i.e., smaller). It is also observed that the shared memory CUDA implementation outperforms the regular CUDA implementation.

Table 2. Grading Policy: activities and points

Problem Size NxN	CPU Time (Sec)	GPU Time (Sec)	
		No Shared Memory	Shared Memory
100 x 100	2.47	3.86	3.88
500 x 500	421.35	7.60	6.08
1000 x 1000	1572.87	19.17	11.63
2500 x 2500	6592.91	66.72	34.36
5000 x 5000	12071.26	116.71	50.02

The speedup due to CUDA/GPU implementation over CPU implementation is calculated, as shown in Figure 4. For small problems (100 x 100 in the experiment), the speedup is less than 1.0. However, the speedup increases as the problem size increases. It should be noted that the speedup of CUDA without shared memory is always smaller than that of CUDA with shared memory. It should also be noted that after the problem size exceeds a limit (2500x2500 in this experiment),

although the speedup of CUDA without shared memory is negligible, the speedup of CUDA with shared memory is significant. For problem size 5000x5000 in the experiment, CUDA with shared memory implementation helps reduce processing time from 12071 seconds to 117 seconds (i.e., a speed up factor of 241). It is projected that speed up factor can be increased linerly as the problem size increases by using GPU shared memory.

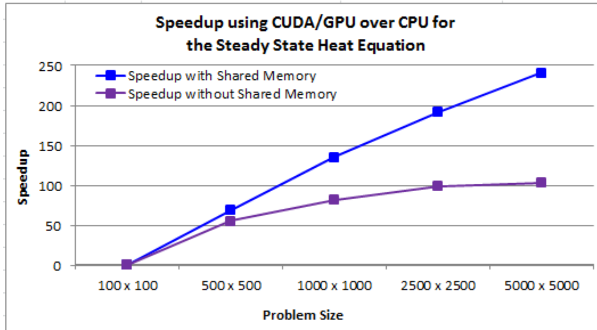

Fig 4: Speedup of discrete heat equation using CUDA/GPU based parallel programming

5.3 Solving Laplace's Equation

In many cases like lightning strikes on a composite material, when the charge distribution is not known, the Poisson's Equation can be used to solve electrostatic problems. Using the Laplacian operator on the electric potential function over a region of the space where the charge density is not zero, the Poisson's Equation is shown below in Equation 4:

$$\nabla^2\varphi = \frac{\partial^2\varphi}{\partial x^2} + \frac{\partial^2\varphi}{\partial y^2} + \frac{\partial^2\varphi}{\partial z^2} = -\frac{\rho}{\epsilon} \quad \text{.......................... (4)}$$

If the charge density is zero all over the region, the Poison's Equation becomes Laplace's equation (see Equation 5). It should be noted that the Laplace's equation is a special case of the steady state heat equation when heat does not vary with time.

$$\nabla^2\varphi = \frac{\partial^2\varphi}{\partial x^2} + \frac{\partial^2\varphi}{\partial y^2} + \frac{\partial^2\varphi}{\partial z^2} = 0 \quad \text{.......................... (5)}$$

Composite mixtures used in electromagnetic meta-material applications often consist of periodic arrangement of unit elements whose sizes are much smaller than the operating wavelength. Therefore, Laplace's equation can be simplified applying the quasi-static assumptions as shown in Equation 6, where ε and φ are the medium permittivity and the electric potential, respectively [25].

$$\nabla \cdot (\varepsilon \nabla \phi) = 0 \quad \text{.. (6)}$$

For very uniform material, electric potential φ can be considered the same in all directions. Now based on the finite-difference approximations, Equation 6 can be rewritten as Equation 7 (a 3D problem that can be solved using the discrete approach).

(φi+1,j,k – φi,j,k)/dx + (φi,j+1,k – φi,j,k)/dy + (φi,j,k+1 – φi,j,k)/dz + (φi,j,k – φi-1,j,k)/dx + (φi,j,k – φi,j-1,k)/dy + (φi,j,k – φi,j,k-1)/dz = 0 .... (7)

Where, dx, dy, and dz are the spatial grid size, the $\varphi_{i,j,k}$ is the electric potential defined at lattice point (i, j, k) and $\varepsilon_x^{i.j.k}$, $\varepsilon_y^{i.j.k}$, and $\varepsilon_z^{i.j.k}$ are the effective x-, y-, and z- direction permittivity defined at edges of the element cell (i, j, k).

The CUDA/C implementation of the 3D charge distribution for GPU is shown in Figure 5. Here, the right values of i and j for each thread is calculated accordingly.

```
/* CUDA/GPU implementation of the charge distribution
equation without shared memory */
__global__ void Charge_Dist_GPU(float *A, float *B, int N) {
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
int k, index, index1, index2, index3, index4, index5, index6;
for (k=1;k<N-1;k++) {
index = k*N*N + j*N + i;
index1=k*N*N + j*N + i+1; index2=k*N*N + j*N + i-1;
index3=k*N*N + (j+1)*N + i; index4=k*N*N + (j-1)*N + i;
index5=(k+1)*N*N + j*N + i; index6=(k-1)*N*N + j*N + i;
if (i>0 && j>0 && k>0 && i<(N-1) && j<(N-1) && k<(N-1)) {
B[index] = (X[index1]*A[index1] + X[index2]*A[index2] +
Y[index3]*A[index3] + Y[index4]*A[index4] +
[index5]*A[index5] + Z[index6]*A[index6]) / 6.0;
}
}/* end Charge_Dist _GPU */
```

Fig 5: Electric Charge Distribution Equation (main loop in CUDA/C without shared memory)

Finally, the shared memory CUDA/C implementation of the 3D heat transfer for GPU is shown in Figure 6. Here, in addition to calculating the right values of i and j for each thread, the shared variables As and Bs are also created.

```
/* CUDA/GPU implementation of the charge distribution
equation with shared memory */
__global__ void Charge_Dist_GPU_SM(float *A, float *B, int N)
{
int i = blockIdx.x * blockDim.x + threadIdx.x;
int j = blockIdx.y * blockDim.y + threadIdx.y;
int is = threadIdx.x ; int js = threadIdx.y;
__shared__ float As[THRDIM][THRDIM];
int ks, index, index1, index2, index3, index4, index5, index6;
As[threadIdx.x][threadIdx.y] = A[index];
__syncthreads();
for (ks=1;ks<N-1;ks++) {
index = ks*N*N + js*N + is;
index1 = ks*N*N + js*N + is+1;
index2 = ks*N*N + js*N + is-1;
index3 = ks*N*N + (js+1)*N + is;
index4 = ks*N*N + (js-1)*N + is;
index5 = (ks+1)*N*N + js*N + is;
index6 = (ks-1)*N*N + js*N + is;
if (is>0 && js>0 && ks>0 && is<(N-1) && js<(N-1) && ks<(N-1))
{
B[index] = (X[index1]*A[index1] + X[index2]*A[index2] +
Y[index3]*A[index3] + Y[index4]*A[index4] +
Z[index5]*A[index5] + Z[index6]*A[index6]) / 6.0;
}
}/* end Charge_Dist_GPU_SM */
```

Fig 6: Electric Charge Distribution Equation (main loop in CUDA/C with shared memory)

The speedup due to CUDA/GPU implementation over CPU implementation for Laplace's equation is shown in Figure 7. Like steady state heat equation, for small problems, the speedup is less than 1.0 and the speedup increases as the problem size increases. Again, the speedup of CUDA without shared memory is always smaller than that of CUDA with shared memory. Also noted that after the problem size exceeds a limit, the speedup of CUDA without shared

memory does not increase, but the speedup of CUDA with shared memory increases significantly.

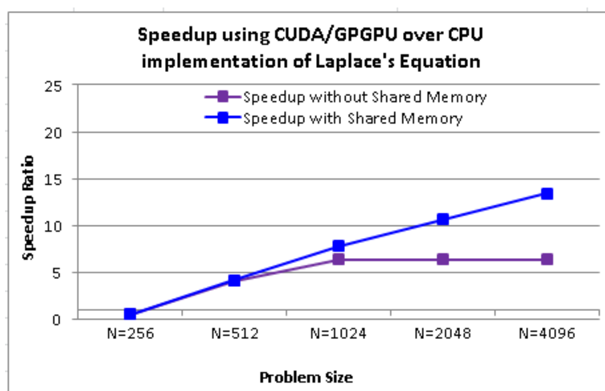

Fig 7: Speedup of Laplace's equation using CUDA/GPU

5.4 Separable Convolution Filter

Separable convolution is a technique for fast convolution [26]. It is commonly used in computer vision, image processing, signal processing, etc. Convolution is a mathematical operation on two functions (say, 'f' and 'g') that produces a third function (say, 'c'). Function 'c' is typically viewed as a modified version of one of the original functions (say, 'f') giving the area overlap between the two functions (as illustrated in Figure 8). In this experiment, CUDA/GPGPU assisted separable convolution filter implementation is introduced.

5.4.1 Separable Filters

A separable filter is a special type of filter that can be expressed as the composition of two 1-D (one dimensional) filters, one on the rows on the image, and one on the columns. For a width n and height m filter kernel, a two-dimensional convolution filter normally requires n*m multiplications for each output pixel. A separable filter can be divided into two consecutive one-dimensional convolution operations on the data, and therefore requires only (n + m) multiplications for each output pixel.

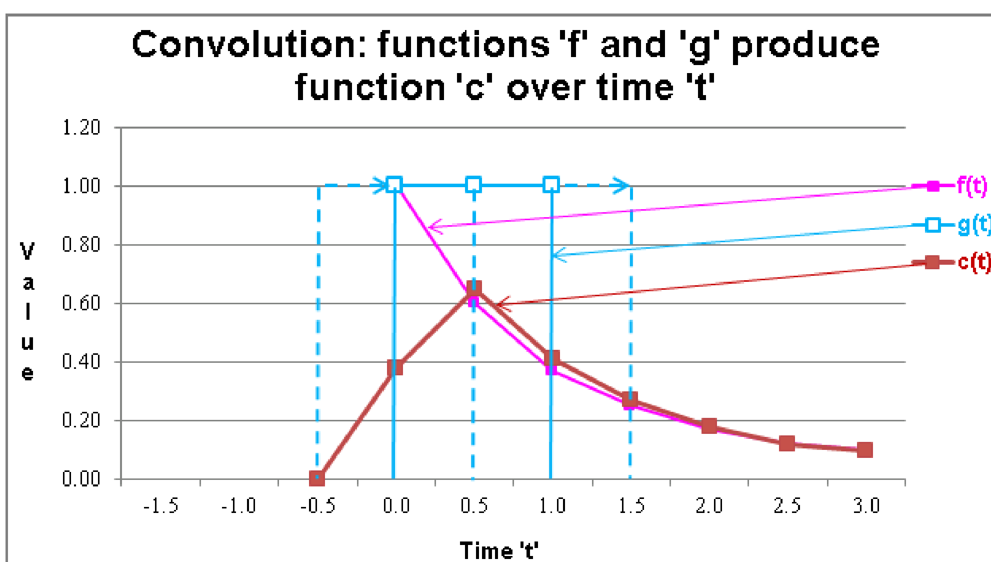

Fig 8: Separable convolution filter – applying function c(t) to some data is the same as applying f(t) followed by g(t)

For example, the 3x3 filter shown below is a separable Sobel [26] edge detection filter. Because applying

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \text{ to the data is the same as applying } \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \text{ followed by } \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}.$$

Separable filters offer more flexibility in the implementation and, in addition, reduction of the arithmetic complexity and bandwidth usage of the computation for each data point.

5.4.2 A Simple CUDA Implementation

According to this approach, (i) a block of the image is loaded into a shared memory array, (ii) a point-wise multiplication of a filter-size portion of the block is done, and (iii) the sum is written into the output image in device memory. Each thread block processes one block in the image. Each thread generates a single output pixel. An illustration of this is shown in Figure 9. To filter the image block, an apron of pixels is required. An apron of pixels is around the image block within a thread block of the width of the kernel radius. The apron of one block overlaps with adjacent blocks and requires special attention (like the threads loading the apron pixels will be idle during the filter computation) to implement properly. Five major steps involved in this approach are: (i) Random input data values are used in the experiment. (ii) Gaussian convolution kernel is calculated and copied to CUDA constant array. As the Gaussian is a symmetric function, the row and column filters are identical. (iii) CUDA computation grid is configured for requested image and filter parameters. (iv) Row and column filters are applied onto the input data. (v) The resulting image is copied back to the CPU and checked for correctness.

Experimental results suggest that significant performance improvement can be achieved due to shared memory CUDA/GPU implementation of separable convolution filters.

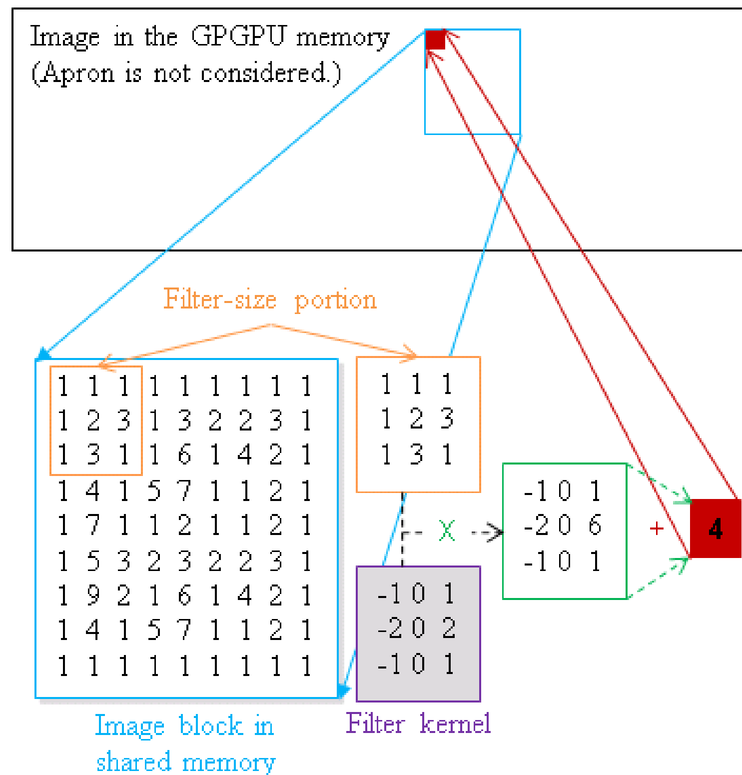

Fig 9: Simple implementation of a separable convolution filter using GPU/CUDA technology

6. CONCLUSION

Parallel computing and multicore computers are today's actuality. Concurrent/parallel processing has the potential to speed up the execution of very complex and large problems. The growing demands for high performance computing can be fulfilled by developing effective parallel programs suitable for multicore/manycore systems. Recent reports show growing demands in parallel programming jobs. Therefore, universities are expected to prepare the new graduates with proper knowledge and skills with parallel thinking. Present computer science and engineering curricula more or less teach the parallel programming APIs like OpenMP, MPI, and CUDA, but do not develop 'think in parallel' skills by addressing the transition from single-core to multicore architecture and sequential to parallel programming.

This paper introduces an effective approach to equip the students with fundamental knowledge and analytic skills to understand large complex problems and develop parallel computing solutions to meet current and future requirements for developing computer applications. As an experiment, multithreaded parallel programming is introduced to undergraduate/graduate level science and engineering students through an existing course. Multicore architecture and multithreaded programming are covered; how to dissect a problem and develop parallel programming for multicore CPU and manycore GPU systems using CUDA/C are taught. In the laboratory, CUDA/GPU assisted parallel programs are developed to solve (i) the Steady State Heat Equation for different 2D thin surfaces, (ii) the Laplace's Equation for electric charge distribution, and (iii) the Convolution for image processing. Experimental results from Steady State Heat Equation show that up to 241x speedup can be achieved for an error tolerance of 0.0001. It is worthy to mention that the parallel solution has potential to save energy consumption by reducing the execution time.

The feedbacks and advices from the director of Wichita State University (WSU) high performance computing center (HiPeCC) and the CEO of M2SYS Technology are greatly appreciated. The Student Outcomes Assessment Reports are also reviewed for this course. It is concluded that the proposed approach has potential to provide adequate knowledge and training so that students should be able to develop parallel programs for complex problems.

7. ACKNOWLEDGMENTS

Mr. John Matrow, Director of WSU HiPeCC, is sincerely acknowledged for his effort to review students' projects and provide valuable advices. The students are also acknowledged for their effort to provide constructive feedbacks.

8. REFERENCES

[1] NVIDIA. 2014. Nvidia: CUDA. http://www.nvidia.com/object/cuda_home_new.html (accessed on Feb 1, 2014).

[2] Udacity. 2014. Introduction to Parallel Programming. https://www.udacity.com/course/cs344 (accessed on Feb 1, 2014).

[3] Intel Developer Zone. 2014. Intel Many Integrated Core Architecture (Intel MIC Architecture). http://software.intel.com/en-us/forums/intel-many-integrated-core (accessed on Feb 1, 2014).

[4] Marowka, A. 2008. Think Parallel: Teaching Parallel Programming Today. IEEE Distributed Systems Online, Vol. 9, No. 8.

[5] Mellor-Crummey, J., Gropp, W., and Herlihy, M. 2010. Teaching parallel programming: a roundtable discussion. XRDS: Crossroads, The ACM Magazine for Students - The Changing Face of Programming, Vol. 17, No. 1, pp. 28-30.

[6] OpenMP. 2014. The OpenMP API specification for parallel programming. http://openmp.org/wp/ (accessed on Feb 1, 2014).

[7] Multicore Programming Education. 2009. Workshop on Directions in Multicore Programming Education. Washington DC.

[8] Multicore LA. 2011. Open Source Software, Multicore and Parallel Computing Miniconference. http:// multicorelca.wordpress.com (accessed on Feb 1, 2014).

[9] Zhu, Y. 2008. Supercomputing Undergraduate Program in Maine (SuperMe). NSF RUE Award 0754951.

[10] Zhang, W. 2011. Collaborative Proposal: Problem-Based Learning of Multithreaded Programming. NSF CCLI Award1063644.

[11] Brown, R. 2010. A strategy for injecting parallel computing education throughout the computer science curriculum. NSF CCLI Award 0942190.

[12] insidePHC. 2014. Trends Show Huge Growth in Parallel Programming Job Market. http://insidehpc.com/ 2011/07/16/trends-show-huge-growth-in-parallel-programming-job-market/ (accessed on Feb 1, 2014).

[13] Asaduzzaman, A., Asmatulu, R., and Pendse, R. 2013. Thinking in Parallel: Multicore Parallel Programming for STEM Education. American Society for Engineering Education (ASEE'13) Midwest Section Annual Conference, Salina, Kansas.

[14] Open MPI. 2014. Open MPI: High Performance Computing. http://www.open-mpi.org/ (accessed on Feb 1, 2014).

[15] Ernst, D.J., et al. 2008. Concurrent CS: Preparing Students for a Multicore World. ITiCSE'08, 2008.

[16] Adams, J., Nevison, C. and Schaller, N.C. 2000. Parallel computing to start the millennium. Proceedings of the thirty-first SIGCSE technical symposium on Computer science education, ACM publication, Vol. 32 Issue 1, pp. 65-69.

[17] Alaghband, G. and Jordan, H.F. 1994. Overview of the force scientific parallel language. Journal Scientific Programming, Vol. 3, No. 1.

[18] Amato, N.M., Iyer, R., Sundaresan, S., and Wu, Y. 1996. A Comparison of Parallel Sorting Algorithms on Different Architectures. Technical Report 98-029, Department of Computer Science, Texas A&M University.

[19] Blelloch, G.E., Leiserson, C.E., Maggs, B.M., Plaxton, C.G., Smith, S.J., and Zagha, M. 1991. A comparison of sorting algorithms for the Connection Machine CM-2. Annual ACM symposium on parallel algorithms and architectures, pp. 3–16.

[20] Li, H. and Sevcik, K.C. 1994. Parallel sorting by over partitioning. Proceedings of the sixth annual ACM symposium on parallel algorithms and architectures (SPAA'94), pp. 46–56.

[21] Rivest, R.L., Shamir, A., and Adleman, L.M. 1977. RSA algorithm. U.S. Patent 4,405,829.

[22] CUDA. 2014. http://en.wikipedia.org/wiki/CUDA (accessed on Feb 1, 2014).

[23] The 2D/3D heat equation. 2014. www.maths.bris.ac.uk/ ~marp/apde2/week3notes.pdf (accessed on Feb 1, 2014).

[24] Asaduzzaman, A., Yip, C.M., Kumar, S., and Asmatulu, R. 2013. Fast, Effective, and Adaptable Computer Modelling and Simulation of Lightning Strike Protection on Composite Materials. IEEE SoutheastCon Conference 2013, Jacksonville, Florida.

[25] Lightning Strike Protection for Carbon Fiber Aircraft. 2914. White paper, Dexmet Corporation. URL: http://www.dexmet.com/1_pdf/LSP%20for%20Carbon %20Fiber%20Aircraft.pdf (accessed on Feb 1, 2014).

[26] An Introduction to Edge Detection: The Sobel Edge Detector. 2014. Generation5. http://www.generation5. org/content/2002/im01.asp (accessed on Feb 1, 2014).